

## SEMESTRE 2 / COURS 1 - PREUVES & COMPLEXITÉ

---

ITC MPSI & PCSI – Année 2025-2026



1. Introduction
2. Terminaison
3. Correction
4. Complexité

# INTRODUCTION

---

Un algorithme est une suite finie de règles et d'opérations élémentaires mises en œuvre sur un nombre fini de données en vue de résoudre un problème spécifique. Plusieurs questions se posent.

Un algorithme est une suite finie de règles et d'opérations élémentaires mises en œuvre sur un nombre fini de données en vue de résoudre un problème spécifique. Plusieurs questions se posent.

- Un algorithme se termine-t-il ?

Un algorithme est une suite finie de règles et d'opérations élémentaires mises en œuvre sur un nombre fini de données en vue de résoudre un problème spécifique. Plusieurs questions se posent.

- Un algorithme se termine-t-il ?  
→ *Terminaison* de l'algorithme

Un algorithme est une suite finie de règles et d'opérations élémentaires mises en œuvre sur un nombre fini de données en vue de résoudre un problème spécifique. Plusieurs questions se posent.

- Un algorithme se termine-t-il ?  
→ *Terminaison* de l'algorithme
- Un algorithme répond-il aux attentes ?

Un algorithme est une suite finie de règles et d'opérations élémentaires mises en œuvre sur un nombre fini de données en vue de résoudre un problème spécifique. Plusieurs questions se posent.

- Un algorithme se termine-t-il ?  
→ *Terminaison* de l'algorithme
- Un algorithme répond-il aux attentes ?  
→ *Correction* de l'algorithme



Un algorithme est une suite finie de règles et d'opérations élémentaires mises en œuvre sur un nombre fini de données en vue de résoudre un problème spécifique. Plusieurs questions se posent.

- Un algorithme se termine-t-il ?  
→ *Terminaison* de l'algorithme
- Un algorithme répond-il aux attentes ?  
→ *Correction* de l'algorithme
- Combien de temps et de ressources mémoires ?

Un algorithme est une suite finie de règles et d'opérations élémentaires mises en œuvre sur un nombre fini de données en vue de résoudre un problème spécifique. Plusieurs questions se posent.

- Un algorithme se termine-t-il ?  
→ *Terminaison* de l'algorithme
- Un algorithme répond-il aux attentes ?  
→ *Correction* de l'algorithme
- Combien de temps et de ressources mémoires ?  
→ *Complexité* de l'algorithme

# OBJECTIFS

- Établir la terminaison d'un programme simple.

# OBJECTIFS

- Établir la terminaison d'un programme simple.
- Établir sa correction à l'aide d'un invariant de boucle.

- Établir la terminaison d'un programme simple.
- Établir sa correction à l'aide d'un invariant de boucle.
- Évaluer la complexité temporelle et/ou spatiale d'un programme.

- Établir la terminaison d'un programme simple.
- Établir sa correction à l'aide d'un invariant de boucle.
- Évaluer la complexité temporelle et/ou spatiale d'un programme.

Preuve = Terminaison + Correction

# TERMINAISON

---



# ANALYSE D'UNE BOUCLE

```
n = 5  
while n != 0 :  
    n -= 1
```

# ANALYSE D'UNE BOUCLE

```
n = 5
while n != 0 :
    n -= 1
```

- Valeur initiale de `n` : 5.

# ANALYSE D'UNE BOUCLE

```
n = 5
while n != 0 :
    n -= 1
```

- Valeur initiale de **n** : 5.
- Valeurs successives de **n** : 4, 3, 2, 1, 0.

# ANALYSE D'UNE BOUCLE

```
n = 5
while n != 0 :
    n -= 1
```

- Valeur initiale de **n** : 5.
- Valeurs successives de **n** : 4, 3, 2, 1, 0.
- Condition **n != 0** non vérifiée quand **n = 0**.

# ANALYSE D'UNE BOUCLE

```
n = 5
while n != 0 :
    n -= 1
```

- Valeur initiale de **n** : 5.
- Valeurs successives de **n** : 4, 3, 2, 1, 0.
- Condition **n != 0** non vérifiée quand **n = 0**.

La boucle s'arrête.

# ANALYSE D'UNE BOUCLE

```
n = 5
while n != 0 :
    n -= 1
```

- Valeur initiale de **n** : 5.
- Valeurs successives de **n** : 4, 3, 2, 1, 0.
- Condition **n != 0** non vérifiée quand **n = 0**.

La boucle s'arrête.

Le nombre d'opérations effectuées par ce programme est fini : le programme *termine*.

## ANALYSE D'UNE BOUCLE

```
n = -1  
while n != 0 :  
    n -= 1
```

# ANALYSE D'UNE BOUCLE

```
n = -1  
while n != 0 :  
    n -= 1
```

- Valeur initiale de **n** : -1.



# ANALYSE D'UNE BOUCLE

```
n = -1
while n != 0 :
    n -= 1
```

- Valeur initiale de **n** : -1.
- Valeurs successives de **n** : -2, -3, -4, etc.

# ANALYSE D'UNE BOUCLE

```
n = -1
while n != 0 :
    n -= 1
```

- Valeur initiale de **n** : -1.
- Valeurs successives de **n** : -2, -3, -4, etc.
- Condition **n != 0** toujours vérifiée.

# ANALYSE D'UNE BOUCLE

```
n = -1
while n != 0 :
    n -= 1
```

- Valeur initiale de **n** : -1.
- Valeurs successives de **n** : -2, -3, -4, etc.
- Condition **n != 0** toujours vérifiée.

La boucle ne s'arrête jamais.

# ANALYSE D'UNE BOUCLE

```
n = -1  
while n != 0 :  
    n -= 1
```

- Valeur initiale de **n** : -1.
- Valeurs successives de **n** : -2, -3, -4, etc.
- Condition **n != 0** toujours vérifiée.

La boucle ne s'arrête jamais.

Le nombre d'opérations effectuées par ce programme est infini : le programme ne *termine* pas.

L'algorithme précédent termine si  $n$  est initialement un entier positif,

L'algorithme précédent termine si  $n$  est initialement un entier positif, c'est donc une pré-condition possible pour la boucle.

L'algorithme précédent termine si  $n$  est initialement un entier positif, c'est donc une pré-condition possible pour la boucle.

Nous avons vu dans un précédent chapitre que cette pré-condition peut-être testée à l'aide d'une assertion avant la boucle :

L'algorithme précédent termine si  $n$  est initialement un entier positif, c'est donc une pré-condition possible pour la boucle.

Nous avons vu dans un précédent chapitre que cette pré-condition peut-être testée à l'aide d'une assertion avant la boucle :

```
assert n >= 0, "la valeur de n doit être positive ou \
↪ nulle"
while n != 0:
    n -= 1
```



La notation **var**<sub>*i*</sub> désigne le contenu de la variable **var** à la fin de l'exécution de la *i*<sup>e</sup> itération de la boucle.

La notation  $\mathbf{var}_i$  désigne le contenu de la variable  $\mathbf{var}$  à la fin de l'exécution de la  $i^{\text{e}}$  itération de la boucle.

Par convention  $\mathbf{var}_0$  désigne le contenu de la variable juste avant d'entrer dans la boucle.

# NOTATION

La notation  $\mathbf{var}_i$  désigne le contenu de la variable  $\mathbf{var}$  à la fin de l'exécution de la  $i^{\text{e}}$  itération de la boucle.

Par convention  $\mathbf{var}_0$  désigne le contenu de la variable juste avant d'entrer dans la boucle.

## Exemple

```
S = 10
for k in range(2, 5):
    S = S + k
```

# NOTATION

La notation  $\mathbf{var}_i$  désigne le contenu de la variable  $\mathbf{var}$  à la fin de l'exécution de la  $i^{\text{e}}$  itération de la boucle.

Par convention  $\mathbf{var}_0$  désigne le contenu de la variable juste avant d'entrer dans la boucle.

## Exemple

```
S = 10
for k in range(2, 5):
    S = S + k
```

$S_0 = 10, S_1 = 12, S_2 = 15, S_3 = 19.$

## Résultat principal pour la terminaison

Si  $a$  est un réel et  $(u_i)$  une suite d'entiers strictement croissante (resp. strictement décroissante), alors il existe un rang  $i_0$  pour lequel  $u_{i_0} > a$  (resp.  $u_{i_0} < a$ ).

## Résultat principal pour la terminaison

Si  $a$  est un réel et  $(u_i)$  une suite d'entiers strictement croissante (resp. strictement décroissante), alors il existe un rang  $i_0$  pour lequel  $u_{i_0} > a$  (resp.  $u_{i_0} < a$ ).

## Méthodologie pour établir une terminaison

Exhiber une suite positive, dépendant des données du programme, à valeurs dans  $\mathbb{N}$ , qui décroît strictement à chaque passage dans la boucle.

## EXEMPLE

Calcul de  $x^n$  - algorithme d'exponentiation rapide.

```
def expR(x: float, n: int)->float :  
    """ Renvoie  $x^n$  pour  $x$  réel et  $n$  entier naturel. \  
    ↪ """  
    X, N, R = x, n, 1  
    while N != 0 :  
        if N%2 == 0 :  
            N = N//2  
        else :  
            R = R*X  
            N = (N-1)//2  
    X = X*X  
    return R
```

## EXEMPLE

```
def expR(x: float, n: int)->float :  
    """ Renvoie  $x^n$  pour  $x$  réel et  $n$  entier naturel. """  
    X, N, R = x, n, 1  
    while N != 0 :  
        if N%2 == 0 :  
            N = N//2  
        else :  
            R = R*X  
            N = (N-1)//2  
    X = X*X  
    return R
```

- **Hypothèse** : la boucle ne se termine jamais. Soit  $N_i$  la valeur de **N** à la fin de l'itération  $i$ , donc :  $\forall i \in \mathbb{N}, N_i \neq 0$ .



## EXEMPLE

```
def expR(x: float, n: int)->float :  
    """ Renvoie  $x^n$  pour  $x$  réel et  $n$  entier naturel. """  
    X, N, R = x, n, 1  
    while N != 0 :  
        if N%2 == 0 :  
            N = N//2  
        else :  
            R = R*X  
            N = (N-1)//2  
    X = X*X  
    return R
```

- **Hypothèse** : la boucle ne se termine jamais. Soit  $N_i$  la valeur de **N** à la fin de l'itération  $i$ , donc :  $\forall i \in \mathbb{N}, N_i \neq 0$ .
- Valeur initiale :  $N_0 = n \in \mathbb{N}$  (pré-condition).
- Supposons pour un entier  $i$ , que  $N_i \in \mathbb{N}$  :

## EXEMPLE

```
def expR(x: float, n: int)->float :  
    """ Renvoie  $x^n$  pour  $x$  réel et  $n$  entier naturel. """  
    X, N, R = x, n, 1  
    while N != 0 :  
        if N%2 == 0 :  
            N = N//2  
        else :  
            R = R*X  
            N = (N-1)//2  
    X = X*X  
    return R
```

- **Hypothèse** : la boucle ne se termine jamais. Soit  $N_i$  la valeur de **N** à la fin de l'itération  $i$ , donc :  $\forall i \in \mathbb{N}, N_i \neq 0$ .
- Valeur initiale :  $N_0 = n \in \mathbb{N}$  (pré-condition).
- Supposons pour un entier  $i$ , que  $N_i \in \mathbb{N}$  :
  1. si  $N_i$  est pair alors  $N_{i+1} = N_i/2 \in \mathbb{N}$  et  $N_{i+1} < N_i$  (car  $N_i \neq 0$ !), donc  $0 < N_{i+1} < N_i$ ;

## EXEMPLE

```
def expR(x: float, n: int)->float :  
    """ Renvoie  $x^n$  pour  $x$  réel et  $n$  entier naturel. """  
    X, N, R = x, n, 1  
    while N != 0 :  
        if N%2 == 0 :  
            N = N//2  
        else :  
            R = R*X  
            N = (N-1)//2  
    X = X*X  
    return R
```

- **Hypothèse** : la boucle ne se termine jamais. Soit  $N_i$  la valeur de **N** à la fin de l'itération  $i$ , donc :  $\forall i \in \mathbb{N}, N_i \neq 0$ .
- Valeur initiale :  $N_0 = n \in \mathbb{N}$  (pré-condition).
- Supposons pour un entier  $i$ , que  $N_i \in \mathbb{N}$  :
  1. si  $N_i$  est pair alors  $N_{i+1} = N_i/2 \in \mathbb{N}$  et  $N_{i+1} < N_i$  (car  $N_i \neq 0$ !), donc  $0 < N_{i+1} < N_i$ ;
  2. si  $N_i$  est impair alors  $N_{i+1} = (N_i - 1)/2 \in \mathbb{N}$  et  $N_{i+1} < N_i$ , donc  $0 < N_{i+1} < N_i$ ;

## EXEMPLE

```
def expR(x: float, n: int) -> float :  
    """ Renvoie  $x^n$  pour  $x$  réel et  $n$  entier naturel. """  
    X, N, R = x, n, 1  
    while N != 0 :  
        if N % 2 == 0 :  
            N = N // 2  
        else :  
            R = R * X  
            N = (N - 1) // 2  
    X = X * X  
    return R
```

- **Hypothèse** : la boucle ne se termine jamais. Soit  $N_i$  la valeur de **N** à la fin de l'itération  $i$ , donc :  $\forall i \in \mathbb{N}, N_i \neq 0$ .
- Valeur initiale :  $N_0 = n \in \mathbb{N}$  (pré-condition).
- Supposons pour un entier  $i$ , que  $N_i \in \mathbb{N}$  :
  1. si  $N_i$  est pair alors  $N_{i+1} = N_i / 2 \in \mathbb{N}$  et  $N_{i+1} < N_i$  (car  $N_i \neq 0$ !), donc  $0 < N_{i+1} < N_i$ ;
  2. si  $N_i$  est impair alors  $N_{i+1} = (N_i - 1) / 2 \in \mathbb{N}$  et  $N_{i+1} < N_i$ , donc  $0 < N_{i+1} < N_i$ ;

## EXAMPLE

```
def expR(x: float, n: int) -> float :  
    """ Renvoie  $x^n$  pour  $x$  réel et  $n$  entier naturel. """  
    X, N, R = x, n, 1  
    while N != 0 :  
        if N % 2 == 0 :  
            N = N // 2  
        else :  
            R = R * X  
            N = (N - 1) // 2  
    X = X * X  
    return R
```

### Bilan

Pour tout entier naturel  $i$  :

- $N_i \in \mathbb{N}$ ;

## EXEMPLE

```
def expR(x: float, n: int) -> float :  
    """ Renvoie  $x^n$  pour  $x$  réel et  $n$  entier naturel. """  
    X, N, R = x, n, 1  
    while N != 0 :  
        if N % 2 == 0 :  
            N = N // 2  
        else :  
            R = R * X  
            N = (N - 1) // 2  
    X = X * X  
    return R
```

### Bilan

Pour tout entier naturel  $i$  :

- $N_i \in \mathbb{N}$ ;
- la suite  $(N_i)$  est strictement décroissante.

## EXEMPLE

```
def expR(x: float, n: int) -> float :  
    """ Renvoie  $x^n$  pour  $x$  réel et  $n$  entier naturel. """  
    X, N, R = x, n, 1  
    while N != 0 :  
        if N % 2 == 0 :  
            N = N // 2  
        else :  
            R = R * X  
            N = (N - 1) // 2  
    X = X * X  
    return R
```

### Bilan

Pour tout entier naturel  $i$  :

- $N_i \in \mathbb{N}$ ;
- la suite  $(N_i)$  est strictement décroissante.

Ceci est absurde, donc la boucle **while** se termine.

## EXERCICE

Prouver la terminaison du code suivant :

```
def f(x: float, n: int)->float :  
    y = 1  
    i = 0  
    while i < n :  
        y *= x  
        i += 1  
    return y
```

- **Hypothèse** : On suppose que la boucle ne termine pas, on a donc :

$$\forall k \in \mathbb{N}, \quad i_k < n.$$



## EXERCICE

Prouver la terminaison du code suivant :

```
def f(x: float, n: int)->float :  
    y = 1  
    i = 0  
    while i < n :  
        y *= x  
        i += 1  
    return y
```

- **Hypothèse** : On suppose que la boucle ne termine pas, on a donc :

$$\forall k \in \mathbb{N}, \quad i_k < n.$$

- On a  $i_0 = 0$ ,  $i_{k+1} = i_k + 1 > i_k$ ,  $(i_k)$  suite d'entiers strictement croissante,

## EXERCICE

Prouver la terminaison du code suivant :

```
def f(x: float, n: int)->float :  
    y = 1  
    i = 0  
    while i < n :  
        y *= x  
        i += 1  
    return y
```

- **Hypothèse** : On suppose que la boucle ne termine pas, on a donc :

$$\forall k \in \mathbb{N}, \quad i_k < n.$$

- On a  $i_0 = 0$ ,  $i_{k+1} = i_k + 1 > i_k$ ,  $(i_k)$  suite d'entiers strictement croissante,
- $\exists k_0, i_{k_0} \geq n$ , d'où contradiction avec l'hypothèse : la boucle termine.

- Soit la suite  $u_0 = n$ , où  $n$  est un entier naturel, et vérifiant

$$\forall p \in \mathbb{N} \quad u_{p+1} = \begin{cases} u_p/2 & \text{si } u_p \text{ est pair;} \\ 3u_p + 1 & \text{si } u_p \text{ est impair.} \end{cases}$$

## UN EXEMPLE ATYPIQUE : LA SUITE DE SYRACUSE

- Soit la suite  $u_0 = n$ , où  $n$  est un entier naturel, et vérifiant

$$\forall p \in \mathbb{N} \quad u_{p+1} = \begin{cases} u_p // 2 & \text{si } u_p \text{ est pair;} \\ 3u_p + 1 & \text{si } u_p \text{ est impair.} \end{cases}$$

- Conjecture (non démontrée) : quelque soit  $n$ , il existe un rang à partir duquel la suite prend les valeurs 4, 2, 1 de manière périodique.

## UN EXEMPLE ATYPIQUE : LA SUITE DE SYRACUSE

- La fonction suivante termine-t-elle?

```
def syracuse(n: int)->int:
    """ Calcule les termes de la suite de Syracuse
    commençant par l'entier strictement positif n
    jusqu'à ce qu'un terme vaille 1 et renvoie
    l'indice de ce dernier """
    u = n
    while u != 1 :
        if u%2 == 0 :
            u = u//2
        else :
            u = 3*u+1
    return 1
```

## UN EXEMPLE ATYPIQUE : LA SUITE DE SYRACUSE

- La fonction suivante termine-t-elle?

```
def syracuse(n: int)->int:
    """ Calcule les termes de la suite de Syracuse
    commençant par l'entier strictement positif n
    jusqu'à ce qu'un terme vaille 1 et renvoie
    l'indice de ce dernier """
    u = n
    while u != 1 :
        if u%2 == 0 :
            u = u//2
        else :
            u = 3*u+1
    return 1
```

- Oui en théorie, mais on ne sait pas le prouver.

## CORRECTION

---

## Objectif

Établir qu'à l'issue d'une boucle **for** ou **while**, on obtient bien le résultat attendu.



## Objectif

Établir qu'à l'issue d'une boucle **for** ou **while**, on obtient bien le résultat attendu.

## Méthode

Spécifier et démontrer, par récurrence, des « invariants de boucle ».

## Objectif

Établir qu'à l'issue d'une boucle **for** ou **while**, on obtient bien le résultat attendu.

## Méthode

Spécifier et démontrer, par récurrence, des « invariants de boucle ».

- *Un invariant de boucle est une propriété qui dépend des données de l'algorithme et qui est vérifiée à chaque passage dans la boucle.*

## Objectif

Établir qu'à l'issue d'une boucle **for** ou **while**, on obtient bien le résultat attendu.

## Méthode

Spécifier et démontrer, par récurrence, des « invariants de boucle ».

- *Un invariant de boucle est une propriété qui dépend des données de l'algorithme et qui est vérifiée à chaque passage dans la boucle.*
- *On cherche un invariant de boucle qui entraîne le résultat attendu après la dernière itération.*

## Définition

- ensemble  $L = \{l_0, l_1, \dots, l_{n-1}\}$  de  $n$  valeurs,

## Définition

- ensemble  $L = \{l_0, l_1, \dots, l_{n-1}\}$  de  $n$  valeurs,
- moyenne :  $m = \frac{1}{n} \sum_{k=0}^{n-1} l_k$

# MOYENNE D'UNE LISTE DE NOMBRES

## Définition

- ensemble  $L = \{l_0, l_1, \dots, l_{n-1}\}$  de  $n$  valeurs,
- moyenne :  $m = \frac{1}{n} \sum_{k=0}^{n-1} l_k$

## ■ Moyenne des éléments d'une liste

```
def moyenne(L: list)->float :  
    """ Calcule la moyenne des éléments de la liste \  
    ↪ de nombres L """  
    S = 0  
    #  
    # à compléter  
    #  
    return S/len(L)
```

# MOYENNE D'UNE LISTE DE NOMBRES

## Définition

- ensemble  $L = \{l_0, l_1, \dots, l_{n-1}\}$  de  $n$  valeurs,
- moyenne :  $m = \frac{1}{n} \sum_{k=0}^{n-1} l_k$

## ■ ■ Moyenne des éléments d'une liste

```
def moyenne(L: list)->list :  
    """ Calcule la moyenne des éléments de la liste \  
    ↪ de nombres L """  
    S = 0  
    for e in L :  
        S = S+e  
    return S/len(L)
```

# MOYENNE D'UNE LISTE DE NOMBRES : CORRECTION

## ■ Moyenne des éléments d'une liste

```
def moyenne(L: list)->float :  
    """ Calcule la moyenne des éléments de la liste \  
    ↪ de nombres L """  
    S = 0  
    for e in L :  
        S = S+e  
    return S/len(L)
```

### Invariant

Montrer que la propriété suivante est un invariant de boucle :

$$\forall i \in \{0, \dots, n\}, \ll S_i = \sum_{k=0}^{i-1} L[k] \gg$$



## MOYENNE D'UNE LISTE DE NOMBRES : CORRECTION

Comme cela est conseillé, on écrit l'invariant dans le code sous forme de commentaires :

### ■ ■ Moyenne des éléments d'une liste

```
def moyenne(L: list)->float :  
    """ Calcule la moyenne des éléments de la liste \  
    ↪ de nombres L """  
    S = 0  
    for e in L :  
        # Invariant:  $S_i = L[0] + \dots + L[i-1]$   
        S = S+e  
    return S/len(L)
```

# MOYENNE D'UNE LISTE DE NOMBRES : CORRECTION

## Invariant de boucle

$\forall i \in \{0, \dots, n\}, \quad \ll S_i = \sum_{k=0}^{i-1} L[k] \gg \quad \text{où } i \text{ désigne le numéro de l'itération.}$

# MOYENNE D'UNE LISTE DE NOMBRES : CORRECTION

## Invariant de boucle

$\forall i \in \{0, \dots, n\}$ , «  $S_i = \sum_{k=0}^{i-1} L[k]$  » où  $i$  désigne le numéro de l'itération.

## Initialisation

Par convention, si  $a > b$ , on convient que  $\sum_{k=a}^b u_k = 0$ . Ainsi, pour  $i = 0$ , on a bien :  $S_0 = 0 = \sum_{k=0}^{-1} L[k]$ .

# MOYENNE D'UNE LISTE DE NOMBRES : CORRECTION

## Invariant de boucle

$\forall i \in \{0, \dots, n\}$ , «  $S_i = \sum_{k=0}^{i-1} L[k]$  » où  $i$  désigne le numéro de l'itération.

## Initialisation

Par convention, si  $a > b$ , on convient que  $\sum_{k=a}^b u_k = 0$ . Ainsi, pour  $i = 0$ , on a bien :  $S_0 = 0 = \sum_{k=0}^{-1} L[k]$ .

## Hérédité

Supposons la propriété vraie en un certain rang  $i \in \{0, \dots, n-1\}$ . Lors de la  $(i+1)^{\text{e}}$  itération de la boucle, la variable **e** contient  $L[i]$ . L'instruction **S += e** exécutée conduit à :

$$S_{i+1} = S_i + L[i] = \sum_{k=0}^{i-1} L[k] + L[i] = \sum_{k=0}^i L[k]. \quad \text{Ce qui achève la récurrence.}$$

### Correction de la boucle

Comme la dernière itération de la boucle a lieu pour  $i = n$ , la valeur renvoyée par la fonction est bien :  $S_n = \sum_{k=0}^{n-1} L[k]$  et la dernière ligne renvoie bien la moyenne des nombres contenus dans la liste **L**.

## Évaluation d'un polynôme

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_2 x^2 + a_1 x + a_0$$

les coefficients sont stockés dans une liste de longueur  $n + 1$  :

$$LC = [a_n, a_{n-1}, \dots, a_2, a_1, a_0].$$

# ALGORITHME DE HORNER

## Évaluation d'un polynôme

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_2 x^2 + a_1 x + a_0$$

les coefficients sont stockés dans une liste de longueur  $n + 1$  :

$$LC = [a_n, a_{n-1}, \dots, a_2, a_1, a_0].$$

## Algorithme de Horner

```
def evalP(x: float, LC: list)->float:
    """ Évalue  $P(x)$  où  $P$  est le polynôme dont les \
    ↪ coefficients sont \
    dans la liste LC dans l'ordre décroissant des \
    ↪ degrés. """
    P = 0
    for c in LC:
        P = P * x + c
    return(P)
```

## Invariant de boucle

$n + 1$  est la longueur de la liste **LC** :

$$\forall i \in \{0, \dots, n + 1\}, \ll P_i = \sum_{k=0}^{i-1} LC[k] \times x^{i-1-k} \gg.$$



## Invariant de boucle

$n + 1$  est la longueur de la liste **LC** :

$$\forall i \in \{0, \dots, n + 1\}, \ll P_i = \sum_{k=0}^{i-1} LC[k] \times x^{i-1-k} \gg.$$

## Initialisation

Pour  $i = 0$ , on a bien  $P_0 = 0 = \sum_{k=0}^{-1} LC[k] \times x^{-1-k}$ .

# ALGORITHME DE HORNER

## Invariant de boucle

$n + 1$  est la longueur de la liste **LC** :

$$\forall i \in \{0, \dots, n + 1\}, \ll P_i = \sum_{k=0}^{i-1} LC[k] \times x^{i-1-k} \gg.$$

## Initialisation

Pour  $i = 0$ , on a bien  $P_0 = 0 = \sum_{k=0}^{-1} LC[k] \times x^{-1-k}$ .

## Hérédité

Supposons la propriété vraie en un certain rang  $i \in \{0, \dots, n\}$ . Lors de la  $(i + 1)^{\text{e}}$  itération la variable  $c$  contient  $LC[i]$ , donc l'instruction **P = P\*x+c** conduit à :

$$P_{i+1} = P_i \times x + LC[i] = \left( \sum_{k=0}^{i-1} LC[k] \times x^{i-1-k} \right) \times x + LC[i] = \sum_{k=0}^i LC[k] \times x^{i-k}.$$

## Correction de la boucle

Dernière itération pour  $i = n + 1$ , d'après l'invariant la valeur renvoyée est :

$$\begin{aligned} P_{n+1} &= \sum_{k=0}^n LC[k] \times x^{n-k} \\ &= LC[0] \times x^n + LC[1] \times x^{n-1} + \dots + LC[n-1] \times x + LC[n] \\ &= \boxed{a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0} \end{aligned}$$

si on note  $LC = [a_n, a_{n-1}, \dots, a_1, a_0]$ .

# ALGORITHME DE HORNER

```
def evalP(x: float, LC: list)->float:
    """ Évalue  $P(x)$  où  $P$  est le polynôme dont les \
    ↪ coefficients sont
        dans la liste LC dans l'ordre décroissant des \
    ↪ degrés. """
    P = 0
    for c in LC:
        # Invariant :  $P_i = \sum_{k=0}^{i-1} LC[k] \times x^{i-1-k}$  \
        ↪
        P = P * x + c
    return(P)
```

## EXEMPLE BOUCLE `while` : EXPONENTIATION RAPIDE

Calcul de  $x^n$  - algorithme d'exponentiation rapide.

```
def expR(x: float, n: int) -> float :
```

```
    """ Calcule  $x^n$  par la méthode de l'exponentiation rapide pour  $x$  réel et  $n$  entier naturel. |
```

```
    ↪ """
```

```
    X, N, R = x, n, 1
```

```
    while N != 0 :
```

```
        if N%2 == 0 :
```

```
            N = N//2
```

```
        else :
```

```
            R = R*X
```

```
            N = (N-1)//2
```

```
    X = X*X
```

```
    return R
```

### Terminaison

- Pour une boucle `while`, on commence à établir la terminaison de la boucle.

## EXEMPLE BOUCLE `while` : EXPONENTIATION RAPIDE

Calcul de  $x^n$  - algorithme d'exponentiation rapide.

```
def expR(x: float, n: int) -> float :
```

```
    """ Calcule  $x^n$  par la méthode de l'exponentiation rapide pour  $x$  réel et  $n$  entier naturel. |
```

```
    ↪ """
```

```
    X, N, R = x, n, 1
```

```
    while N != 0 :
```

```
        if N%2 == 0 :
```

```
            N = N//2
```

```
        else :
```

```
            R = R*X
```

```
            N = (N-1)//2
```

```
            X = X*X
```

```
    return R
```

### Terminaison

- Pour une boucle `while`, on commence à établir la terminaison de la boucle.
- Ce résultat a déjà été établi plus haut.

### Invariant de boucle

Montrons l'invariant de boucle, où  $\ell$  désigne le nombre d'itérations :

$$\forall i \in \{0, \dots, \ell\}, \text{ «} R_i \times X_i^{N_i} = x^n \text{»}.$$

## EXEMPLE BOUCLE `while` : EXPONENTIATION RAPIDE

### Invariant de boucle

Montrons l'invariant de boucle, où  $\ell$  désigne le nombre d'itérations :

$$\forall i \in \{0, \dots, \ell\}, \ll R_i \times X_i^{N_i} = x^n \gg.$$

### Initialisation

Pour  $i = 0$ , on a  $R_0 \times X_0^{N_0} = 1 \times x^n = x^n$ .



### Hérédité

Supposons la propriété vraie en un certain rang  $i \in \{0, \dots, \ell - 1\}$ . Lors de la  $(i + 1)^{\text{e}}$  itération de la boucle `while` :

## EXEMPLE BOUCLE **while** : EXPONENTIATION RAPIDE

### Hérédité

Supposons la propriété vraie en un certain rang  $i \in \{0, \dots, \ell - 1\}$ . Lors de la  $(i + 1)^{\text{e}}$  itération de la boucle **while** :

1<sup>er</sup> **cas** : si  $N_i$  est pair, alors les exécutions de  $N = N//2$  et  $X = X * X$  conduisent à  $R_{i+1} = R_i$ ,  $N_{i+1} = N_i/2$  et  $X_{i+1} = X_i^2$  donc :

$$R_{i+1} \times X_{i+1}^{N_{i+1}} = R_i \times (X_i^2)^{\frac{N_i}{2}} = R_i \times X_i^{N_i} = x^n.$$

## EXEMPLE BOUCLE **while** : EXPONENTIATION RAPIDE

### Hérédité

Supposons la propriété vraie en un certain rang  $i \in \{0, \dots, \ell - 1\}$ . Lors de la  $(i + 1)^{\text{e}}$  itération de la boucle **while** :

1<sup>er</sup> **cas** : si  $N_i$  est pair, alors les exécutions de  $N = N//2$  et  $X = X*X$  conduisent à  $R_{i+1} = R_i$ ,  $N_{i+1} = N_i/2$  et  $X_{i+1} = X_i^2$  donc :

$$R_{i+1} \times X_{i+1}^{N_{i+1}} = R_i \times (X_i^2)^{\frac{N_i}{2}} = R_i \times X_i^{N_i} = x^n.$$

2<sup>e</sup> **cas** : si  $N_i$  est impair, alors les exécutions de  $R=R*X$ ,  $N=(N-1)//2$  et  $X=X*X$  conduisent à  $R_{i+1} = R_i \times X_i$ ,  $N_{i+1} = (N_i - 1)/2$  et  $X_{i+1} = X_i^2$  donc :

$$R_{i+1} \times X_{i+1}^{N_{i+1}} = R_i \times X_i \times (X_i^2)^{\frac{N_i-1}{2}} = R_i \times X_i^{N_i} = x^n.$$

## EXEMPLE BOUCLE **while** : EXPONENTIATION RAPIDE

### Correction de la boucle

La dernière itération de la boucle **while** a lieu pour  $i = \ell$ , et on a  $N_\ell = 0$  car il n'y a pas d'itération  $\ell + 1$ .

La valeur renvoyée par la fonction est bien :

$$R_\ell = R_\ell \times \underbrace{X_\ell^{N_\ell}}_{=1} = \boxed{X^n}.$$

## EXEMPLE BOUCLE `while` : EXPONENTIATION RAPIDE

```
def expR(x: float, n: int)->float :  
    """ Calcule  $x^n$  par la méthode de \  
    ↪ l'exponentiation rapide  
    pour  $x$  réel et  $n$  entier naturel. """  
    X, N, R = x, n, 1  
    while N != 0 :  
        # invariant :  $R_i \times X_i^{N_i} = x^n$   
        if N%2 == 0 :  
            N = N//2  
        else :  
            R = R*X  
            N = (N-1)//2  
        X = X*X  
    return R
```

## EXERCICE

### Partie entière

Pour  $x$  réel positif, la partie entière de  $x$  est le plus grand entier naturel inférieur ou égal à  $x$ . La fonction suivante effectue le calcul :

```
def ParEnt(x: float)->int :  
    """ Calcule la partie entière du réel positif x. \  
    ↪ """  
    n = 0  
    while n+1 <= x :  
        n += 1  
    return n
```

Faire la preuve de cette fonction.

## ET AVEC UNE FONCTION RÉCURSIVE ?

Considérons la fonction :

```
def f(a: int, b: int)->int :  
    """ Calcul récursif du pgcd, a et b sont \  
    ↪ supposés naturels """  
    if b == 0:  
        return a  
    else:  
        return f(b, a%b)
```

On peut établir la terminaison et la correction en montrant par récurrence sur le paramètre  $b$  :

## ET AVEC UNE FONCTION RÉCURSIVE ?

Considérons la fonction :

```
def f(a: int, b: int)->int :  
    """ Calcul récursif du pgcd, a et b sont \  
    ↪ supposés naturels """  
    if b == 0:  
        return a  
    else:  
        return f(b, a%b)
```

On peut établir la terminaison et la correction en montrant par récurrence sur le paramètre  $b$  :

$P(b) : \forall a \in \mathbb{N}, f(a, b)$  se termine et renvoie  $\text{pgcd}(a, b)$ .



## ET AVEC UNE FONCTION RÉCURSIVE ?

- Initialisation.

## ET AVEC UNE FONCTION RÉCURSIVE ?

- **Initialisation.** Il est clair que  $P(0)$  est vrai (c'est le cas terminal et  $\text{pgcd}(a, 0) = a$ ).

## ET AVEC UNE FONCTION RÉCURSIVE ?

- **Initialisation.** Il est clair que  $P(0)$  est vrai (c'est le cas terminal et  $\text{pgcd}(a, 0) = a$ ).
- **Hérédité.**

## ET AVEC UNE FONCTION RÉCURSIVE ?

- **Initialisation.** Il est clair que  $P(0)$  est vrai (c'est le cas terminal et  $\text{pgcd}(a, 0) = a$ ).
- **Hérédité.** Supposons la propriété vraie pour tous les entiers jusqu'à un naturel  $b$ , et soit  $a \in \mathbb{N}$ .

## ET AVEC UNE FONCTION RÉCURSIVE ?

- **Initialisation.** Il est clair que  $P(0)$  est vrai (c'est le cas terminal et  $\text{pgcd}(a, 0) = a$ ).
- **Hérédité.** Supposons la propriété vraie pour tous les entiers jusqu'à un naturel  $b$ , et soit  $a \in \mathbb{N}$ .  
Lorsqu'on appelle  $f(a, b + 1)$ , comme  $b + 1 \neq 0$ , on renvoie la valeur de  $f(b + 1, r)$  où  $r$  est le reste de la division de  $a$  par  $b + 1$  :  
$$a = (b + 1)q + r,$$

## ET AVEC UNE FONCTION RÉCURSIVE ?

- **Initialisation.** Il est clair que  $P(0)$  est vrai (c'est le cas terminal et  $\text{pgcd}(a, 0) = a$ ).
- **Hérédité.** Supposons la propriété vraie pour tous les entiers jusqu'à un naturel  $b$ , et soit  $a \in \mathbb{N}$ .  
Lorsqu'on appelle  $f(a, b + 1)$ , comme  $b + 1 \neq 0$ , on renvoie la valeur de  $f(b + 1, r)$  où  $r$  est le reste de la division de  $a$  par  $b + 1$  :  
 $a = (b + 1)q + r$ , comme  $0 \leq r \leq b$ , on sait par hypothèse que  $f(b + 1, r)$  se termine et renvoie  $\text{pgcd}(b + 1, r)$ ,

## ET AVEC UNE FONCTION RÉCURSIVE ?

- **Initialisation.** Il est clair que  $P(0)$  est vrai (c'est le cas terminal et  $\text{pgcd}(a, 0) = a$ ).
- **Hérédité.** Supposons la propriété vraie pour tous les entiers jusqu'à un naturel  $b$ , et soit  $a \in \mathbb{N}$ .

Lorsqu'on appelle  $f(a, b + 1)$ , comme  $b + 1 \neq 0$ , on renvoie la valeur de  $f(b + 1, r)$  où  $r$  est le reste de la division de  $a$  par  $b + 1$  :

$a = (b + 1)q + r$ , comme  $0 \leq r \leq b$ , on sait par hypothèse que  $f(b + 1, r)$  se termine et renvoie  $\text{pgcd}(b + 1, r)$ , donc  $f(a, b + 1)$  se termine et renvoie  $\text{pgcd}(b + 1, r)$ ,

## ET AVEC UNE FONCTION RÉCURSIVE ?

- **Initialisation.** Il est clair que  $P(0)$  est vrai (c'est le cas terminal et  $\text{pgcd}(a, 0) = a$ ).
- **Hérédité.** Supposons la propriété vraie pour tous les entiers jusqu'à un naturel  $b$ , et soit  $a \in \mathbb{N}$ .

Lorsqu'on appelle  $f(a, b + 1)$ , comme  $b + 1 \neq 0$ , on renvoie la valeur de  $f(b + 1, r)$  où  $r$  est le reste de la division de  $a$  par  $b + 1$  :

$a = (b + 1)q + r$ , comme  $0 \leq r \leq b$ , on sait par hypothèse que  $f(b + 1, r)$  se termine et renvoie  $\text{pgcd}(b + 1, r)$ , donc  $f(a, b + 1)$  se termine et renvoie  $\text{pgcd}(b + 1, r)$ , or d'après le cours de mathématique,  $\text{pgcd}(a, b + 1) = \text{pgcd}(b + 1, r)$ ,



## ET AVEC UNE FONCTION RÉCURSIVE ?

- **Initialisation.** Il est clair que  $P(0)$  est vrai (c'est le cas terminal et  $\text{pgcd}(a, 0) = a$ ).
- **Hérédité.** Supposons la propriété vraie pour tous les entiers jusqu'à un naturel  $b$ , et soit  $a \in \mathbb{N}$ .

Lorsqu'on appelle  $f(a, b + 1)$ , comme  $b + 1 \neq 0$ , on renvoie la valeur de  $f(b + 1, r)$  où  $r$  est le reste de la division de  $a$  par  $b + 1$  :

$a = (b + 1)q + r$ , comme  $0 \leq r \leq b$ , on sait par hypothèse que  $f(b + 1, r)$  se termine et renvoie  $\text{pgcd}(b + 1, r)$ , donc  $f(a, b + 1)$  se termine et renvoie  $\text{pgcd}(b + 1, r)$ , or d'après le cours de mathématique,  $\text{pgcd}(a, b + 1) = \text{pgcd}(b + 1, r)$ , donc  $P(b + 1)$  est vraie, ce qui termine la récurrence.

# TOUT N'EST PAS SI SIMPLE !

La fonction suivante :

```
def decompPremier(p: int)-> (int,int):  
    """ Renvoie deux entiers u et v tels que  $p = u^2 + v^2$  p doit être un nombre premier congru à 1 modulo 4 """  
    ↪ 4 """  
  
def f(a: int, b: int, c: int)->(int, int, int):  
    """ La fonction magique locale """  
    if a > b+c:  
        return (a-b-c, b, 2*b+c)  
    else:  
        return (b+c-a, a, 2*a-c)  
  
a, b, c = (p-1)//4, 1, 1  
while a != b:  
    a, b, c = f(a, b, c)  
return (2*a,c)
```

permet de décomposer tout nombre premier congru à 1 modulo 4 en somme de deux carrés.

# TOUT N'EST PAS SI SIMPLE !

La fonction suivante :

```
def decompPremier(p: int)-> (int,int):  
    """ Renvoie deux entiers u et v tels que  $p = u^2 + v^2$  p doit être un nombre premier congru à 1 modulo 4  
    ↪ 4 """  
  
    def f(a: int, b: int, c: int)->(int, int, int):  
        """ La fonction magique locale """  
        if a > b+c:  
            return (a-b-c, b, 2*b+c)  
        else:  
            return (b+c-a, a, 2*a-c)  
  
    a, b, c = (p-1)//4, 1, 1  
    while a != b:  
        a, b, c = f(a, b, c)  
    return (2*a,c)
```

permet de décomposer tout nombre premier congru à 1 modulo 4 en somme de deux carrés.

Un théorème dit que c'est toujours possible. Par exemple,  
`decompPremier(601)` renvoie (24, 5) et on a bien  $601 = 24^2 + 5^2$ .

# TOUT N'EST PAS SI SIMPLE !

La fonction suivante :

```
def decompPremier(p: int)-> (int,int):  
    """ Renvoie deux entiers u et v tels que  $p = u^2 + v^2$  p doit être un nombre premier congru à 1 modulo 4 """  
    ↪ 4 """  
  
    def f(a: int, b: int, c: int)->(int, int, int):  
        """ La fonction magique locale """  
        if a > b+c:  
            return (a-b-c, b, 2*b+c)  
        else:  
            return (b+c-a, a, 2*a-c)  
  
    a, b, c = (p-1)//4, 1, 1  
    while a != b:  
        a, b, c = f(a, b, c)  
    return (2*a,c)
```

permet de décomposer tout nombre premier congru à 1 modulo 4 en somme de deux carrés.

Un théorème dit que c'est toujours possible. Par exemple,

`decompPremier(601)` renvoie (24, 5) et on a bien  $601 = 24^2 + 5^2$ .

Mais la terminaison et la preuve de cet algorithme sont vraiment difficiles.

# COMPLEXITÉ

---

## Objectif

Mesurer « l'efficacité » d'un programme en terme de temps de calcul, en fonction de la taille des données, indépendamment de la puissance d'exécution de l'ordinateur sur lequel le programme est exécuté.

## Objectif

Mesurer « l'efficacité » d'un programme en terme de temps de calcul, en fonction de la taille des données, indépendamment de la puissance d'exécution de l'ordinateur sur lequel le programme est exécuté.

## Méthode

Déterminer un entier  $n$  mesurant la « taille » des données du programme, et compter le nombre  $C(n)$  « d'opérations élémentaires » (à préciser suivant le contexte) nécessaires à l'exécution du programme.

Classer cette complexité en utilisant la relation de domination «  $O$  » entre suites fournissant une « ordre de grandeur » simplifié de la complexité.

## EXEMPLE : FONCTION MOYENNE

fonction

```
def moyenne(L:list)->float :  
    """Calcule la moyenne des éléments  
    de la liste de nombres L"""  
    S = 0  
    for e in L :  
        S += e  
    return S/len(L)
```



## EXEMPLE : FONCTION MOYENNE

fonction

```
def moyenne(L:list)->float :  
    """Calcule la moyenne des éléments  
    de la liste de nombres L"""  
    S = 0  
    for e in L :  
        S += e  
    return S/len(L)
```

complexité

- une affectation avant la boucle **for**,

## EXEMPLE : FONCTION MOYENNE

fonction

```
def moyenne(L:list)->float :  
    """Calcule la moyenne des éléments  
    de la liste de nombres L"""  
    S = 0  
    for e in L :  
        S += e  
    return S/len(L)
```

complexité

- une affectation avant la boucle **for**,
- dans la boucle **for** : une addition et une affectation, répétées  $n$  fois,

## EXEMPLE : FONCTION MOYENNE

fonction

```
def moyenne(L:list)->float :  
    """Calcule la moyenne des éléments  
    de la liste de nombres L"""  
    S = 0  
    for e in L :  
        S += e  
    return S/len(L)
```

complexité

- une affectation avant la boucle **for**,
- dans la boucle **for** : une addition et une affectation, répétées  $n$  fois,
- une division dans la dernière ligne.

## EXEMPLE : FONCTION MOYENNE

fonction

```
def moyenne(L:list)->float :  
    """Calcule la moyenne des éléments  
    de la liste de nombres L"""  
    S = 0  
    for e in L :  
        S += e  
    return S/len(L)
```

complexité

- une affectation avant la boucle **for**,
- dans la boucle **for** : une addition et une affectation, répétées  $n$  fois,
- une division dans la dernière ligne.

On a

$$C(n) = 1 + (1 + 1) \times n + 1 = \boxed{2n + 2}$$

## EXERCICE : COMPLEXITÉ D'UNE PREMIÈRE FONCTION

fonction

```
def f1(n:int)->int:  
    x = 0  
    for i in range(n):  
        for j in range(n):  
            x = x+1  
    return x
```

## EXERCICE : COMPLEXITÉ D'UNE PREMIÈRE FONCTION

fonction

```
def f1(n:int)->int:
    x = 0
    for i in range(n):
        for j in range(n):
            x = x+1
    return x
```

complexité

- une affectation avant la première boucle **for**,

## EXERCICE : COMPLEXITÉ D'UNE PREMIÈRE FONCTION

fonction

```
def f1(n:int)->int:
    x = 0
    for i in range(n):
        for j in range(n):
            x = x+1
    return x
```

complexité

- une affectation avant la première boucle **for**,
- dans la boucle **for** en la variable **j** : une addition et une affectation, répétées  $n$  fois, soit  $2n$  opérations élémentaires

## EXERCICE : COMPLEXITÉ D'UNE PREMIÈRE FONCTION

fonction

```
def f1(n:int)->int:
    x = 0
    for i in range(n):
        for j in range(n):
            x = x+1
    return x
```

complexité

- une affectation avant la première boucle **for**,
- dans la boucle **for** en la variable **j** : une addition et une affectation, répétées  $n$  fois, soit  $2n$  opérations élémentaires
- la boucle **for** en la variable **i**, on répète  $n$  fois la boucle en la variable **j**



## EXERCICE : COMPLEXITÉ D'UNE PREMIÈRE FONCTION

fonction

```
def f1(n:int)->int:
    x = 0
    for i in range(n):
        for j in range(n):
            x = x+1
    return x
```

complexité

- une affectation avant la première boucle **for**,
- dans la boucle **for** en la variable **j** : une addition et une affectation, répétées  $n$  fois, soit  $2n$  opérations élémentaires
- la boucle **for** en la variable **i**, on répète  $n$  fois la boucle en la variable **j**

On a

$$C(n) = 1 + n \times (2n) = \boxed{1 + 2n^2}$$

## EXERCICE : COMPLEXITÉ D'UNE SECONDE FONCTION

fonction

```
def f2(n:int)->int:  
    x = 0  
    for i in range(n):  
        for j in range(i):  
            x += 1  
    return x
```

## EXERCICE : COMPLEXITÉ D'UNE SECONDE FONCTION

fonction

```
def f2(n:int)->int:
    x = 0
    for i in range(n):
        for j in range(i):
            x += 1
    return x
```

complexité

- une affectation avant la première boucle **for**,

## EXERCICE : COMPLEXITÉ D'UNE SECONDE FONCTION

fonction

```
def f2(n:int)->int:
    x = 0
    for i in range(n):
        for j in range(i):
            x += 1
    return x
```

complexité

- une affectation avant la première boucle **for**,
- dans la boucle **for** en la variable **j** : une addition et une affectation, répétées  $i$  fois, soit  $2i$  opérations élémentaires

## EXERCICE : COMPLEXITÉ D'UNE SECONDE FONCTION

fonction

```
def f2(n:int)->int:
    x = 0
    for i in range(n):
        for j in range(i):
            x += 1
    return x
```

complexité

- une affectation avant la première boucle **for**,
- dans la boucle **for** en la variable **j** : une addition et une affectation, répétées  $i$  fois, soit  $2i$  opérations élémentaires
- dans la boucle **for** en la variable **i**, on répète la boucle en la variable **j**

## EXERCICE : COMPLEXITÉ D'UNE SECONDE FONCTION

fonction

```
def f2(n:int)->int:
    x = 0
    for i in range(n):
        for j in range(i):
            x += 1
    return x
```

complexité

- une affectation avant la première boucle **for**,
- dans la boucle **for** en la variable **j** : une addition et une affectation, répétées  $i$  fois, soit  $2i$  opérations élémentaires
- dans la boucle **for** en la variable **i**, on répète la boucle en la variable **j**

On a

$$C(n) = 1 + \sum_{i=0}^{n-1} (2i) = \boxed{1 + (n-1)n}$$

## TEMPS D'EXÉCUTION POUR CERTAINES COMPLEXITÉS USUELLES

Processeur exécutant une opération élémentaire en une nanoseconde.

$n$	10	100	1000	10000	100000
$\ln n$	2 ns	5 ns	7 ns	9 ns	12 ns
$n$	10 ns	0.1 $\mu$ s	1 $\mu$ s	10 $\mu$ s	0.1 ms
$n \ln n$	20 ns	0.5 $\mu$ s	7 $\mu$ s	90 $\mu$ s	1 ms
$n^2$	0.1 $\mu$ s	10 $\mu$ s	1 ms	0.1 s	10 s
$n^3$	1 $\mu$ s	1 ms	1 s	17 h	12 j
$2^n$	1 $\mu$ s	$3 \cdot 10^{13}$ a	...	...	...

## DÉSIGNATION DES COMPLEXITÉS

$O(1)$	complexité constante	$O(n \ln n)$	complexité quasi-linéaire
$O(\ln n)$	complexité logarithmique	$O(n^k)$	complexité polynomiale
$O(n)$	complexité linéaire	$O(2^n)$	complexité exponentielle



## EXERCICE

### Moyennes de CÉSARO d'une liste $[u_0, u_1, \dots, u_n]$

Les moyennes de CÉSARO de la liste  $u = [u_0, u_1, \dots, u_n]$  forment la liste  $v = [v_0, v_1, \dots, v_n]$  définie par :

$$v_0 = \frac{u_0}{1}, \quad v_1 = \frac{u_0 + u_1}{2}, \quad v_n = \frac{u_0 + u_1 + \dots + u_n}{n+1}.$$

## EXERCICE

### Moyennes de CÉSARO d'une liste $[u_0, u_1, \dots, u_n]$

Les moyennes de CÉSARO de la liste  $u = [u_0, u_1, \dots, u_n]$  forment la liste  $v = [v_0, v_1, \dots, v_n]$  définie par :

$$v_0 = \frac{u_0}{1}, \quad v_1 = \frac{u_0+u_1}{2}, \quad v_n = \frac{u_0+u_1+\dots+u_n}{n+1}.$$

```
def cesaro(u: list)->list :  
    """Calcul de la liste de Césaró associée à la \  
    ↪ liste u"""  
    v = []  
    for k in range(len(u)) :  
        m = moyenne(u[:k+1])  
        v += [m]  
    return v
```

- Proposer une fonction `cesaro2` de complexité significativement meilleure.

## EXERCICE

```
def cesaro(u:list)->list :  
    """Calcul de la liste de Césaró associée à la \  
↪ liste u"""  
    v = []  
    for k in range(len(u)) :  
        m = moyenne(u[:k+1])  
        v = v + [m]  
    return(v)
```

Rappelons que la fonction **moyenne** vue précédemment appliquée à une liste de longueur  $k$  a pour complexité  $2k+2$ . On en déduit (ici  $n$  est  $\text{len}(u)-1$ ):

## EXERCICE

```
def cesaro(u:list)->list :  
    """Calcul de la liste de Césaró associée à la \  
    ↪ liste u"""  
    v = []  
    for k in range(len(u)) :  
        m = moyenne(u[:k+1])  
        v = v + [m]  
    return(v)
```

Rappelons que la fonction **moyenne** vue précédemment appliquée à une liste de longueur  $k$  a pour complexité  $2k+2$ . On en déduit (ici  $n$  est  $\text{len}(u)-1$ ):

complexité

$$C(n) = 1 + \sum_{k=0}^n [3 + (2(k+1) + 2)] = \dots = \boxed{8 + 8n + n^2 = O(n^2)}.$$

### Moyennes de CÉSARO d'une liste $[u_0, u_1, \dots, u_n]$

L'algorithme précédent recalcule la somme des  $k + 1$  premières valeurs à chaque appel, ce que l'on peut éviter.

## Moyennes de CÉSARO d'une liste $[u_0, u_1, \dots, u_n]$

L'algorithme précédent recalcule la somme des  $k + 1$  premières valeurs à chaque appel, ce que l'on peut éviter.

```
def cesaro2(u:list)->list :  
    """Calcul de la liste de Césaró associée à la liste u"""  
    v = []  
    S = 0  
    for k in range(len(u)) :  
        S = S + u[k]  
        v = v + [S/(k+1)]  
    return v
```

## Moyennes de CÉSARO d'une liste $[u_0, u_1, \dots, u_n]$

L'algorithme précédent recalcule la somme des  $k + 1$  premières valeurs à chaque appel, ce que l'on peut éviter.

```
def cesaro2(u:list)->list :  
    """Calcul de la liste de Césaro associée à la liste u"""  
    v = []  
    S = 0  
    for k in range(len(u)) :  
        S = S + u[k]  
        v = v + [S/(k+1)]  
    return v
```

Complexité de `cesaro2` :

$$C(n) = 2 + 6(n + 1) = \boxed{8 + 6n = O(n)}.$$

## EXERCICE

```
def compte_it(n:int)->int:
    i = n
    x = 0
    while i > 1:
        i = i//2
        x += 1
    return x
```

1. Prouver la terminaison de cette fonction.
2. Justifier l'existence et l'unicité de  $p$  entier tel que :  $2^p \leq n < 2^{p+1}$ .
3. Donner et prouver un invariant sur  $i_k$  de la forme  $i_k \in [a_k, b_k[$  où  $a_k, b_k$  dépendent de  $k$  et  $p$ .
4. En déduire la complexité temporelle de `compte_it`. Quel est le rôle de  $x$ ?



Nombre d'éléments strictement positifs d'une liste

```
def nbPositifs(liste:list)->float :  
    """Nombre d'éléments positifs d'une liste"""  
    nb = 0  
    for elt in liste:  
        if elt > 0 :  
            nb = nb+1  
    return nb
```

Nombre d'éléments strictement positifs d'une liste

```
def nbPositifs(liste:list)->float :  
    """Nombre d'éléments positifs d'une liste"""  
    nb = 0  
    for elt in liste:  
        if elt > 0 :  
            nb = nb+1  
    return nb
```

Présence d'un test **if** : complexité dans le meilleur et dans le pire des cas.

Nombre d'éléments strictement positifs d'une liste

```
def nbPositifs(liste:list)->float :  
    """Nombre d'éléments positifs d'une liste"""  
    nb = 0  
    for elt in liste:  
        if elt > 0 :  
            nb = nb+1  
    return nb
```

Présence d'un test **if** : complexité dans le meilleur et dans le pire des cas.

- Meilleur des cas, éléments tous négatifs :  $C_{\text{meilleur}}(n) = 1 + n \times 1 = n + 1$

## MEILLEUR ET PIRE DES CAS

Nombre d'éléments strictement positifs d'une liste

```
def nbPositifs(liste:list)->float :  
    """Nombre d'éléments positifs d'une liste"""  
    nb = 0  
    for elt in liste:  
        if elt > 0 :  
            nb = nb+1  
    return nb
```

Présence d'un test **if** : complexité dans le meilleur et dans le pire des cas.

- Meilleur des cas, éléments tous négatifs :  $C_{\text{meilleur}}(n) = 1 + n \times 1 = n + 1$
- Pire des cas, éléments tous positifs :  $C_{\text{pire}}(n) = 1 + n \times 3 = 3n + 1$

# MEILLEUR ET PIRE DES CAS

Nombre d'éléments strictement positifs d'une liste

```
def nbPositifs(liste:list)->float :  
    """Nombre d'éléments positifs d'une liste"""  
    nb = 0  
    for elt in liste:  
        if elt > 0 :  
            nb = nb+1  
    return nb
```

Présence d'un test **if** : complexité dans le meilleur et dans le pire des cas.

- Meilleur des cas, éléments tous négatifs :  $C_{\text{meilleur}}(n) = 1 + n \times 1 = n + 1$
- Pire des cas, éléments tous positifs :  $C_{\text{pire}}(n) = 1 + n \times 3 = 3n + 1$
- Cas général :  $n + 1 \leq C_n \leq 3n + 1$ , donc  $C(n) = O(n)$ .

# COMPLEXITÉ FONCTION RÉCURSIVE

Calcul de  $n!$

```
def FactorielleRec(n:int)->int :  
    if n == 0:  
        return 1  
    else:  
        return n*FactorielleRec(n-1)
```

# COMPLEXITÉ FONCTION RÉCURSIVE

Calcul de  $n!$

```
def FactorielleRec(n:int)->int :  
    if n == 0:  
        return 1  
    else:  
        return n*FactorielleRec(n-1)
```

- On trouve  $C(0) = 1$  (car si  $n = 0$ , on compare juste  $n$  à 0)

# COMPLEXITÉ FONCTION RÉCURSIVE

Calcul de  $n!$

```
def FactorielleRec(n:int)->int :  
    if n == 0:  
        return 1  
    else:  
        return n*FactorielleRec(n-1)
```

- On trouve  $C(0) = 1$  (car si  $n = 0$ , on compare juste  $n$  à 0)
- Si  $n \geq 1$ ,  $C(n) = 2 + C(n - 1)$  (on fait une comparaison à 0, une multiplication puis  $C(n - 1)$  opérations élémentaires en appelant la fonction avec le paramètre  $n - 1$ ).



Calcul de  $n!$

```
def FactorielleRec(n:int)->int :  
    if n == 0:  
        return 1  
    else:  
        return n*FactorielleRec(n-1)
```

- On trouve  $C(0) = 1$  (car si  $n = 0$ , on compare juste  $n$  à 0)
- Si  $n \geq 1$ ,  $C(n) = 2 + C(n - 1)$  (on fait une comparaison à 0, une multiplication puis  $C(n - 1)$  opérations élémentaires en appelant la fonction avec le paramètre  $n - 1$ ).
- On obtient donc une suite arithmétique raison 2, ce qui donne après calculs  $C(n) = 1 + 2n = O(n)$ .

## EXERCICE

Complexité d'une fonction réursive

```
def SuiteU(n:int)->float :  
    if n == 0:  
        return 1  
    else:  
        return 2*SuiteU(n-1)+1/SuiteU(n-1)
```

## EXERCICE

Complexité d'une fonction réursive

```
def SuiteU(n:int)->float :  
    if n == 0:  
        return 1  
    else:  
        return 2*SuiteU(n-1)+1/SuiteU(n-1)
```

- $u_0 = 1$  et  $\forall n \in \mathbb{N}, u_{n+1} = 2u_n + \frac{1}{u_n}$ .

## EXERCICE

Complexité d'une fonction récursive

```
def SuiteU(n:int)->float :  
    if n == 0:  
        return 1  
    else:  
        return 2*SuiteU(n-1)+1/SuiteU(n-1)
```

- $u_0 = 1$  et  $\forall n \in \mathbb{N}, u_{n+1} = 2u_n + \frac{1}{u_n}$ .
- $C(0) = 1$  (seulement un test lorsque  $n = 0$ )

## EXERCICE

Complexité d'une fonction réursive

```
def SuiteU(n:int)->float :  
    if n == 0:  
        return 1  
    else:  
        return 2*SuiteU(n-1)+1/SuiteU(n-1)
```

- $u_0 = 1$  et  $\forall n \in \mathbb{N}, u_{n+1} = 2u_n + \frac{1}{u_n}$ .
- $C(0) = 1$  (seulement un test lorsque  $n = 0$ )
- Si  $n \geq 1$ ,  $C(n) = 4 + C(n-1) + C(n-1)$  (on fait un test, une multiplication, une addition, une division et  $2 \times C(n-1)$  opérations élémentaires en appelant deux fois la fonction avec le paramètre  $n-1$ ).

## EXERCICE

### Complexité d'une fonction récursive

```
def SuiteU(n:int)->float :  
    if n == 0:  
        return 1  
    else:  
        return 2*SuiteU(n-1)+1/SuiteU(n-1)
```

- $u_0 = 1$  et  $\forall n \in \mathbb{N}, u_{n+1} = 2u_n + \frac{1}{u_n}$ .
- $C(0) = 1$  (seulement un test lorsque  $n = 0$ )
- Si  $n \geq 1$ ,  $C(n) = 4 + C(n-1) + C(n-1)$  (on fait un test, une multiplication, une addition, une division et  $2 \times C(n-1)$  opérations élémentaires en appelant deux fois la fonction avec le paramètre  $n-1$ ).
- On obtient donc une suite arithmético-géométrique ce qui donne après calculs  $C(n) = 5 \times 2^n - 4$ .

# EXERCICE

## Complexité d'une fonction récursive : amélioration

La fonction précédente ayant une complexité exponentielle, elle devient inutilisable pour de grandes valeurs de  $n$ . Introduisons la fonction suivante :

```
def SuiteU2(n:int)->float :  
    if n == 0:  
        return 1  
    else:  
        a = SuiteU2(n-1)  
        return 2*a+1/a
```

### Complexité d'une fonction récursive : amélioration

La fonction précédente ayant une complexité exponentielle, elle devient inutilisable pour de grandes valeurs de  $n$ . Introduisons la fonction suivante :

```
def SuiteU2(n:int)->float :  
    if n == 0:  
        return 1  
    else:  
        a = SuiteU2(n-1)  
        return 2*a+1/a
```

- Là encore :  $u_0 = 1$  et :  $\forall n \in \mathbb{N}, u_{n+1} = 2u_n + \frac{1}{u_n}$ .



### Complexité d'une fonction récursive : amélioration

La fonction précédente ayant une complexité exponentielle, elle devient inutilisable pour de grandes valeurs de  $n$ . Introduisons la fonction suivante :

```
def SuiteU2(n:int)->float :  
    if n == 0:  
        return 1  
    else:  
        a = SuiteU2(n-1)  
        return 2*a+1/a
```

- Là encore :  $u_0 = 1$  et :  $\forall n \in \mathbb{N}, u_{n+1} = 2u_n + \frac{1}{u_n}$ .
- $C'(0) = 1$  (seulement un test lorsque  $n = 0$ )

### Complexité d'une fonction récursive : amélioration

La fonction précédente ayant une complexité exponentielle, elle devient inutilisable pour de grandes valeurs de  $n$ . Introduisons la fonction suivante :

```
def SuiteU2(n:int)->float :  
    if n == 0:  
        return 1  
    else:  
        a = SuiteU2(n-1)  
        return 2*a+1/a
```

- Là encore :  $u_0 = 1$  et :  $\forall n \in \mathbb{N}, u_{n+1} = 2u_n + \frac{1}{u_n}$ .
- $C'(0) = 1$  (seulement un test lorsque  $n = 0$ )
- Si  $n \geq 1$ ,  $C'(n) = 5 + C'(n-1)$  (on fait un test, une affectation, une multiplication, une addition, une division et  $C'(n-1)$  opérations élémentaires en appelant la fonction avec le paramètre  $n-1$ ). On obtient donc une suite arithmétique qui donne après calculs  $C'(n) = 5n + 1$ .

### Complexité d'une fonction récursive : amélioration

La fonction précédente ayant une complexité exponentielle, elle devient inutilisable pour de grandes valeurs de  $n$ . Introduisons la fonction suivante :

```
def SuiteU2(n:int)->float :  
    if n == 0:  
        return 1  
    else:  
        a = SuiteU2(n-1)  
        return 2*a+1/a
```

- Là encore :  $u_0 = 1$  et :  $\forall n \in \mathbb{N}, u_{n+1} = 2u_n + \frac{1}{u_n}$ .
- $C'(0) = 1$  (seulement un test lorsque  $n = 0$ )
- Si  $n \geq 1$ ,  $C'(n) = 5 + C'(n-1)$  (on fait un test, une affectation, une multiplication, une addition, une division et  $C'(n-1)$  opérations élémentaires en appelant la fonction avec le paramètre  $n-1$ ). On obtient donc une suite arithmétique qui donne après calculs  $C'(n) = 5n + 1$ .
- On obtient une complexité linéaire : le gain est énorme !

## Objectif

Mesurer « l'occupation mémoire » d'un programme en fonction de la taille des données.

## Objectif

Mesurer « l'occupation mémoire » d'un programme en fonction de la taille des données.

## Méthode

Même principe en comptant le nombre  $C(n)$  « d'entités élémentaires de mémoire » (à préciser suivant le contexte) nécessaires à l'exécution du programme.